IBM WebSphere Developer Technical Journal: Get started with WebSphere Integration Developer

Robert Peterson (rrpeters@us.ibm.com), WebSphere Enablement, Austin, TX, IBM

Summary: Learn how to use IBM® WebSphere® Integration Developer V6 by creating a business process, business state machine, and Java™ component for a simple application that uses a services-oriented architecture.

Date: 07 Dec 2005
Level: Intermediate

From the IBM WebSphere Developer Technical Journal.

Introduction

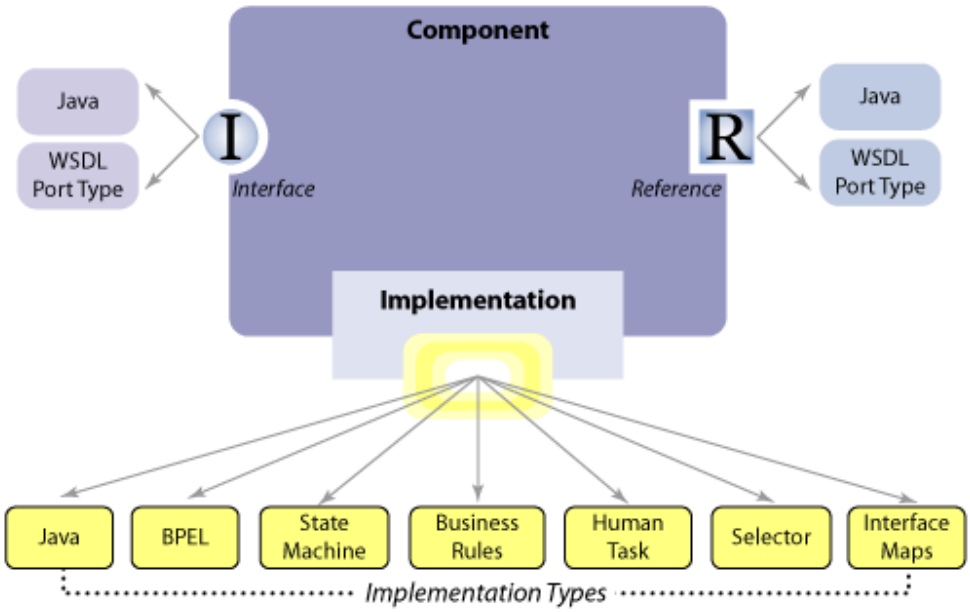
IBM WebSphere Process Server V6, coupled with IBM WebSphere Integration Developer V6, provides a powerful platform for industrial strength applications utilizing a services-oriented architecture. If you are a developer, architect, or administrator that is interested in learning more about this platform, but are unsure of where to get started -- this article is for you. We will walk through an example composed of a servlet that calls a simple stock price application, created with WebSphere Integration Developer.

About the sample application

The sample scenario we will use is a simple stock price service: A stock symbol is sent as a string from a servlet to an application running on IBM WebSphere Process Server. As a result, either a price is returned or an exception is thrown if the stock symbol is invalid. Additionally, the server-side application checks the time of day. If it is before 4:00 PM, an open market price is calculated and returned; otherwise, an after hours price is calculated and returned.

Applications developed with IBM WebSphere Integration Developer utilize a programming model called Service Component Architecture (SCA), a component architecture where each component can use either a WSDL or Java interface completely separate from the implementation.

Figure 1. Service Component Architecture



As shown in Figure 1, there are seven first class SCA component implementations available. The stock price application utilizes three of these components: Java, Business Process (BPEL), and Business State Machine. These components are the richest of the seven because they provide constructs to create business logic. Business Rules, Human Tasks, Selectors, and Interface Maps can include limited business logic, but primarily serve as mapping or routing components. Following are brief descriptions of the components used in the stock price example.

A **business process component** encompasses business logic defined in visual models often referred to as workflow. This component enables real business processes to be modeled with visual components that translate to business process execution language (BPEL). The processes produce BPEL 2.0 which is executed by WebSphere Process Server at run time. The models are intended to be intuitive to a wide audience. It is often the case that business analysts and decision makers work closely with IT architects to create processes that reflect both the business and its IT infrastructure.

A **business state machine component** is very similar in functionality to a business process. It also enables visual components to be wired together to create a model of business logic. A business state machine also produces BPEL 2.0 to be executed at run time. The major difference is that its business logic can be modeled in terms of states (as opposed to actions in a business process). A state can transition to new states, previous states, or itself which is a different paradigm (or way of thinking about business logic).

A **Java component** in SCA represents business logic in conventional Java code which is automatically transacted (similar to an EJB session bean). The implementation is a plain old Java object (POJO) that implements the SCA interface bound to the component. Although any valid Java SE code can be added to a Java component, it is intended for business logic (such as determining which discount a customer should receive). If you are considering a Java component for mapping information, accessing outside systems, or manipulating business objects, look at the other first class SCA components first (see Resources for more information on SCA components). For example, an interface map may be appropriate to map the incoming information from an

external packaged enterprise application.

The above components should be thought of as different styles of implementing business logic that are functionally equivalent. For instance, all three components may be able to implement a procurement procedure, but your organization may find the business process component the most intuitive and beneficial for the task.

Tips for rapid development with WebSphere Integration Developer

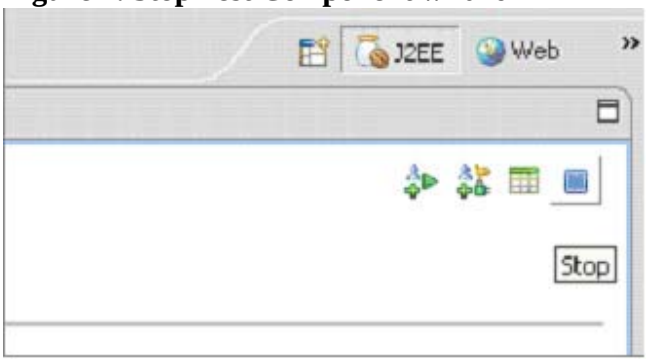
Before developing the stock price application, the following tips for working with WebSphere Integration Developer should prove useful:

- **Save often.** Many of the visual development steps performed in WebSphere Integration Developer translate to various generated artifacts. Pressing **Ctrl-S** periodically to save can enhance development, ensuring that only a few artifacts are created at a time. Note that during the development of the stock price example, errors present themselves in the Problems view that are often corrected in subsequent steps.

- **Clean often.** It is often beneficial to select **Clean...** from the **Projects** menu. This cleans all the project's build artifacts and then automatically performs a rebuild. A clean before deploying to the server or before testing a component is recommended.

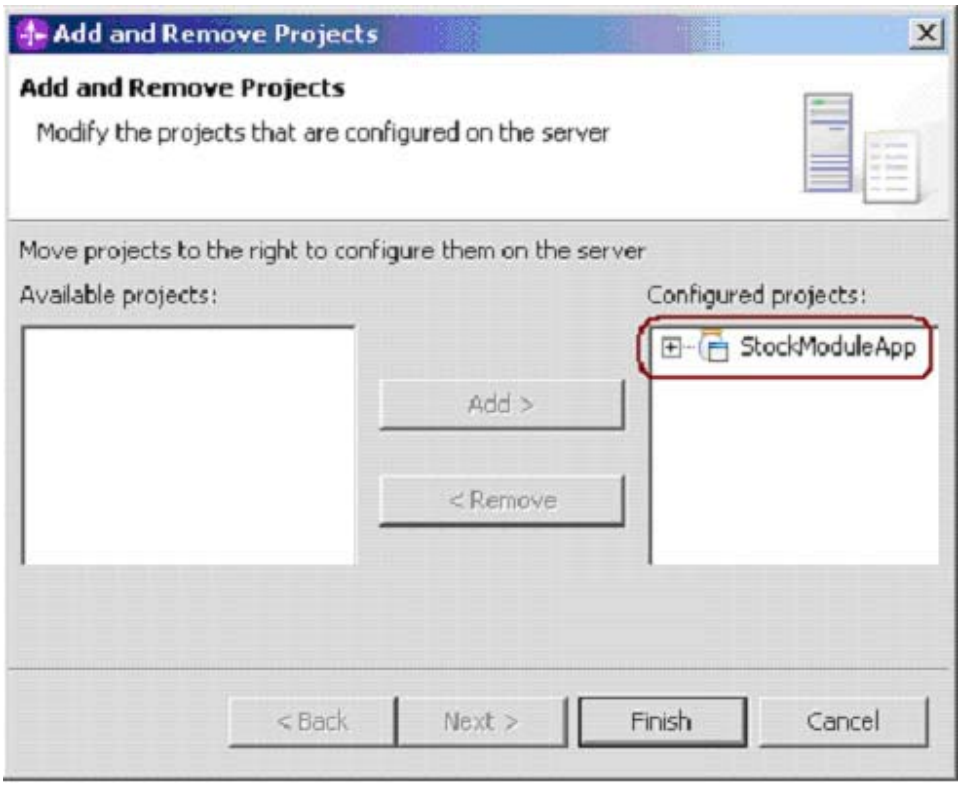
- **Test often.** WebSphere Integration Developer includes a test component wizard that can be used to test SCA components individually. After a component is implemented, right-click the component in the assembly diagram and select **Test Component**. Define the inputs to the component and select **Continue** to unit test the component. Once you are finished with the test wizard, be sure to select the **Stop** button at the top-right (Figure 5).

Figure 2. Stop Test Component wizard



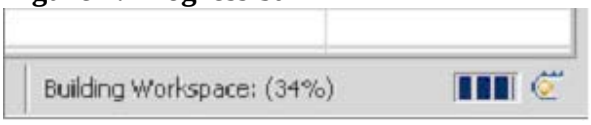
- **One project at a time.** When possible, keep only one project deployed on the integrated WebSphere Process Server. This enables faster server startup and mitigates the risk of conflicting artifacts. This is especially important when developing under different workspaces, because the server only shows the projects deployed in a given workspace. Thus, if projects are not removed from the server before switching to another workspace, they can be forgotten and cause conflicts later (Figure 6).

Figure 3. Project Add/Remove Dialog



- **Watch for the progress bar.** Keep a lookout for the conveyor belt icon at the bottom right with a progress percentage. This means WebSphere Integration Developer is executing a resource intensive task. It is best to wait until the task is complete rather than adding concurrent tasks to its workload.

Figure 4. Progress bar



Build the sample application

The balance of this article walks through the development of the stock price application. The only prerequisite for following these steps is that you have WebSphere Integration Developer V6 installed.

The major steps involved in building the application are:

a. Create an interface in a library
b. Create a module
c. Create and wire the SCA components
d. Implement the Java component
e. Implement the Business State Machine component
f. Implement the Business Process component
g. Create the client servlet
h. Run the servlet from a Web browser

Create an interface in a library

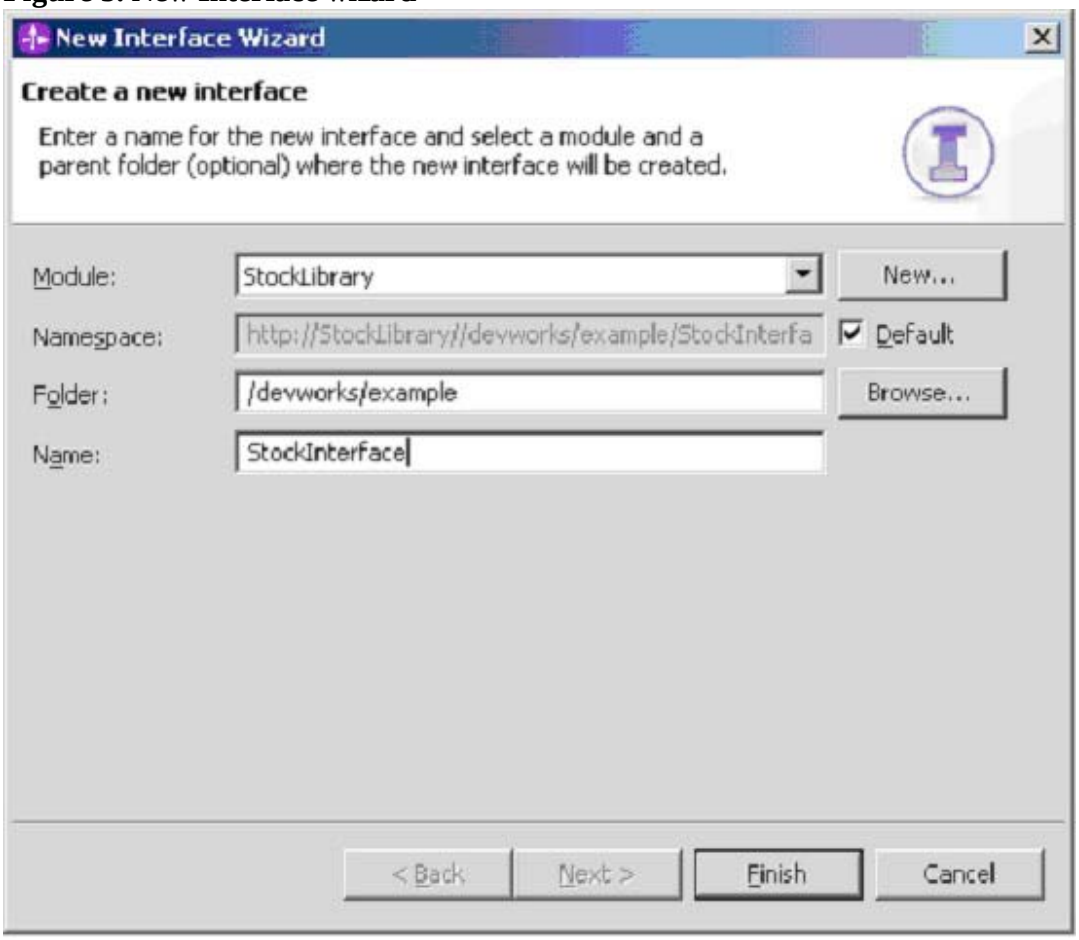
This simple example does not require a library because only one module is used. However, putting commonly used artifacts such as interfaces in a library is often a good practice because they are exposed to all SCA

modules. Thus, this example puts an interface definition it uses, StockInterface, in a library. If a new module is added later to the application, it can use the StockInterface without having to recreate it.

1. In the Business Integration perspective right-click any white space in the left panel and select **New => Library**. Name it `StockLibrary` and select **Finish**.

2. Expand **StockLibrary**. Right-click **Interfaces**, select **New => Interface**, and fill in the wizard as shown in Figure 5.

Figure 5. New Interface wizard



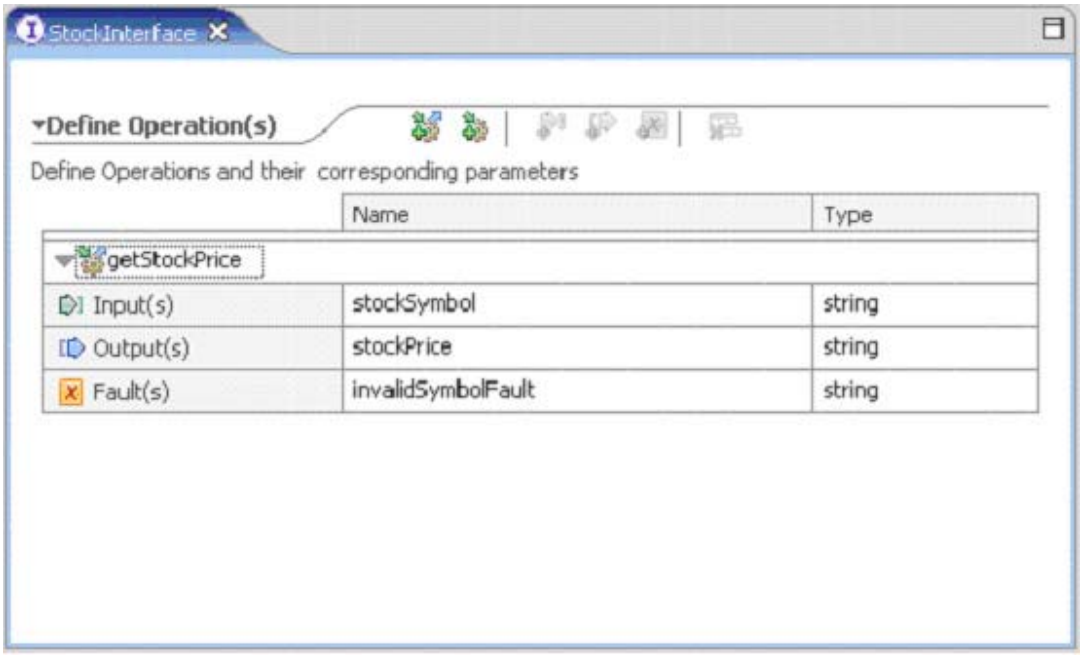
3. In the editor for StockInterface, select the **Add Request Response Operation** icon and rename "operation1" to `getStockPrice`.

4. Select the **Add Input** icon and rename "input1" to `stockSymbol`.

5. Select the **Add Output** icon and rename "output1" to `stockPrice`.

6. Select the **Add Fault** icon and rename "fault1" to `invalidSymbolFault`. The resulting interface definition should be similar to Figure 6.

Figure 6. Stock Interface definition



Create a module

7. Right-click any white space in the Business Integration panel and select **New => Module**. Name it `StockModule` and select **Finish**.

8. We need to import the StockLibrary. Right-click **StockModule** and select **Open Dependency Editor => Add.. => OK**.

Create and wire the SCA components

9. Expand **StockModule** and double-click the nested **StockModule** icon. This should open an editor titled **Assembly Diagram: StockModule**.

10. To add the Business Process component, right-click and select **Add Node => Process**. Rename "Component1" to `StockProcessFacade`.

11. To add the Java component, right-click and select **Add Node=> Java**. Rename "Component1" to `MarketOpenPriceService`.

12. To add the Business State Machine component, right-click and select **Add Node => State Machine**. Rename "Component1" to `MarketClosedPriceService`.

13. All three components use the same interface. Right-click **StockProcessFacade** and select **Add => Interface**. Select **StockInterface** and press **OK**.

14. Repeat the above step for the Java component, Business Process component, and the Business State Machine component.

15. Click and drag the mouse from **StockProcessFacade** (the Business Process component) to the other components to wire them together. Select **Yes** when asked if a new reference should be made.

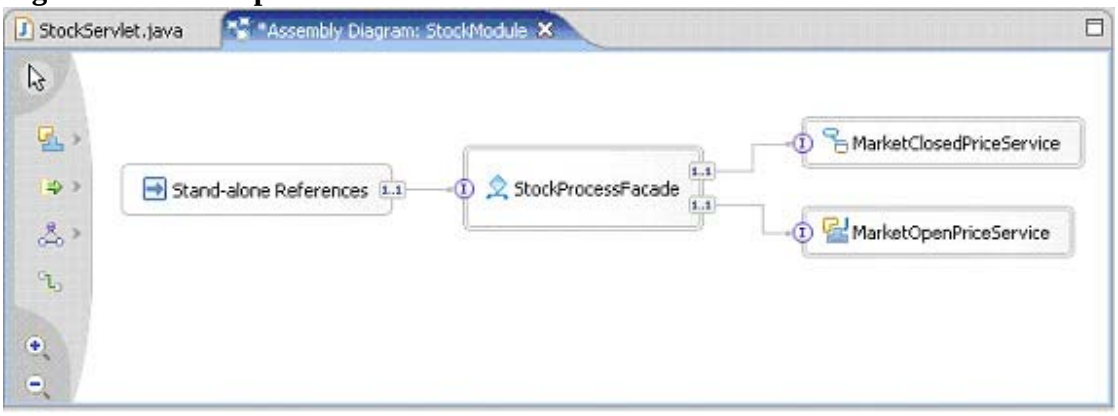
16. External components that a Business Process can invoke are called **Partners**. In this case, MarkedOpenPriceService and MarketClosedPriceService are both partners to StockProcessFacade. Select the small **(1..1)** box on StockProcessFacade that connects to MarketOpenPriceService and select **Properties tab => Details**. Change the name from "StockInterfacePartner" to `MarketOpenPartner`

Similarly, change the name of the "StockInterfacePartner1" to `MarketClosedPartner`.

17. The Business Process component needs to expose a standalone reference for the outside world (in our case a servlet) to invoke it. Right-click the white space in the assembly diagram and select **Add Node => Stand-alone Reference**. Right-click the newly created reference component and select **Add Reference** (or select it from the floating menu). Inspect the defaults; notice that the name of the reference is StockInterfacePartner which is the reference name used in the servlet code. Select **OK**. Select **Yes** when prompted to use a Java interface rather than WSDL.

18. The resulting assembly diagram should be similar to Figure 10.

Figure 7. SCA components



Implement the Java component

The Java SCA component generates a random number up to 200 and returns it as the stock price. In practice, the Java code would access a data store or perhaps an EJB service.

19. In the assembly diagram, right-click the **MarketOpenPriceService** (the Java component) and select **Generate Implementation => OK**.

20. In the Java editor, replace the "//TODO" and return statement inside the getStockPrice method at the bottom of the file with the following:

```
Random r = new Random();
double tmp = r.nextDouble() * 200;
String price = Double.toString(tmp);
return "The active market price is: " + price;
```

21. Press **Shift-Ctrl-O** to organize imports which resolves the java.util.Random reference.

22. Now, test the Java component with the Test Component wizard.

Implement the Business State Machine component

The State Machine component is identical in functionality to the Java component. This is intentional to showcase that first class SCA components in WebSphere Process Server provide different representations and styles of development, but have similar richness in functionality. It should be noted that this simple example does not really showcase the use of a multi-stage state machine. When an application component has many stages that often repeat throughout its lifecycle, a Business State Machine can be a good choice for its implementation.

23. Right-click **MarketClosedPriceService** (the Business State Machine component) and select **Generate Implementation** => **OK** => **New Folder...** and insert `devworks/example/smachine`. Select **OK** and then **OK** again.

24. Notice that the StockInterface was automatically listed at the top right. The first step in creating a state machine is to define a Correlation. WebSphere Process Server uses a unique identifier to reference each instance of the state machine. This state machine will use the stock symbol input as the correlation. Left-click any white space on the editor and then select **Properties tab** => **Correlation**. Select **New** and insert `symbolCorrelation` for the name. Select **Browse** => **string** => **OK** => **New** => **getStockPrice** => **stockSymbol** => **OK**. Finally, select **OK** to exit the Create Correlation wizard.

25. Rename "InititalState1" to `Begin`, "State1" to `no-op`, and "FinalState1" to `End`.

26. The entry point to a Business State Machine is always through an operation of an interface. In this case there is only one interface, StockInterface, and only one operation, called getStockPrice. Select the **operation** icon, **StockInterface** for Interface, and **getStockPrice** for Operation in the Description section.

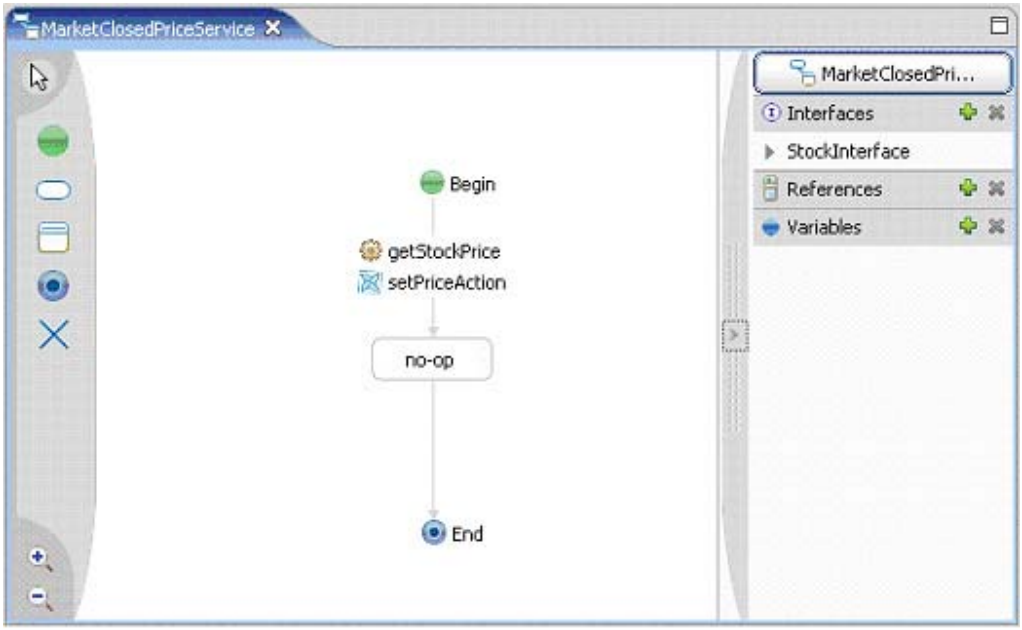
27. When the operation is invoked, the stock price needs to be calculated and bound to the stockPrice property on the getStockPrice operation. A Java Action is used to perform this task. Select the wire between Begin and no-op (not the getStockPrice operation). Select the floating **Add an Action** icon. Rename "Action1" to `setPriceAction`. Select **setPriceAction** => **Properties tab** => **Details** => **Java**. If a window pops up when you select Java, click **Yes**. Insert this code:

```
java.util.Random r = new java.util.Random();
double tmp = r.nextDouble() * 200;
this.getStockPrice_Output_stockPrice =
        "The estimated after hours price is: " +
Double.toString(tmp);
```

28. It should be noted that the setPriceAction is not executed as part of the no-op state. Actions are executed as part of the state above them. Technically, it is executed during the Begin state and, thus, the no-op state is not really needed. States can also have Entry and Exit Java code snippets. The order of execution within a state is Entry code (for initialization), Action code (primary logic), and then Exit code (clean up). (See Resources for more information on Business State Machines.)

29. The resulting Business State Machine should be similar to Figure 8.

Figure 8. Business State Machine



30. Now, test the State Machine component with the Test Component wizard.

Implement the Business Process component

The Business Process component acts as a facade. It checks the current time: if it is before 4:00 pm, it invokes the MarketOpenPriceService Java component; otherwise, it invokes the MarketClosedPriceService Business State Machine component. Additionally, it validates whether the entered stock symbol is composed of only letters and is exactly three characters long. If this is not the case, it throws the invalidSymbolFault defined on StockInterface.

31. In the StockModule Assembly Diagram, right-click **StockProcessFacade** (the Business Process component), select **Generate Implementation** => **New Folder...** and insert `devworks/example/bprocess`. Select **OK** and then **OK** again.

32. A sequence is recommended for a series of actions as it is more efficient than manually wiring consecutive actions together. This business process, however, will not execute a series of actions; therefore, drag **Receive** above the Sequence box and drag **Reply** below it. The sequence should now be empty. Select **Sequence** and delete it.

33. The default variables defined are DataObjects. For simplicity, this business process will use strings (see Resources for more information on Business Objects and Service Data Objects). Rename "InputVariable" to `symbol`. Select **InputVariable** => **Properties tab** => **Details** => **Data Type radio button** => **Browse..** => **string** => **OK**. Rename "OutputVariable" to `price`. Select **OutputVariable** => **Properties tab** => **Details** => **Data Type radio button** => **Browse..** => **string** => **OK**.

34. Receive and Reply need to be configured to bind to the symbol and price variables. Select **Receive** => **Properties tab** => **Details**. Check the **Use Data Type Variables** box and select **(...) button** => **symbol** => **OK**. Select **Reply** => **Properties tab** => **Details**. Check the **Use Data Type Variables** box and select **(...) button** => **price** => **OK**.

35. Next, we will insert a Choice block to determine what action to take based on the symbolValid variable. Right-click **Receive** and select **Add** => **Choice**. Click between the green circle and Receive boxes. Rename "Choice" to `SymbolValidChoice`. Click on the **Case** box and select the **Properties tab** => **Details**=>> **Create a New Condition** => **Java**. Replace the code with the following:

```
if(symbol.length() != 3)
        return false;
for(int i=0; i<symbol.length(); i++) {
        char ch = symbol.charAt(i);
        if(!Character.isLetter(ch)) {
                return false;
        }
}
return true;
```

36. Select **SymbolValidChoice** and then the **Add Otherwise** icon from the floating menu.

37. The otherwise branch is traversed when the stock symbol is not valid, thus a fault should be thrown. Right-click the **otherwise** box and select **Add => Throw**. Rename "Throw" to `ThrowInvalidSymbol`. Select **ThrowInvalidSymbol** and select **Properties tab => Details => User-defined**. Insert `invalidSymbolFault` for the Fault Name field.

38. Another choice must be made based on the time of day: should the MarketOpenPriceService or MarketClosedPriceService be called? Right-click the **Case** box, then select **Add => Choice**. Rename "Choice" to `TimeOfDayChoice`. As before, **Create a New Condition** for the Case box and insert the following code:

```
java.util.Calendar c = java.util.Calendar.getInstance();
final int fourPM = 16;
return c.get(java.util.Calendar.HOUR_OF_DAY) < fourPM;
```

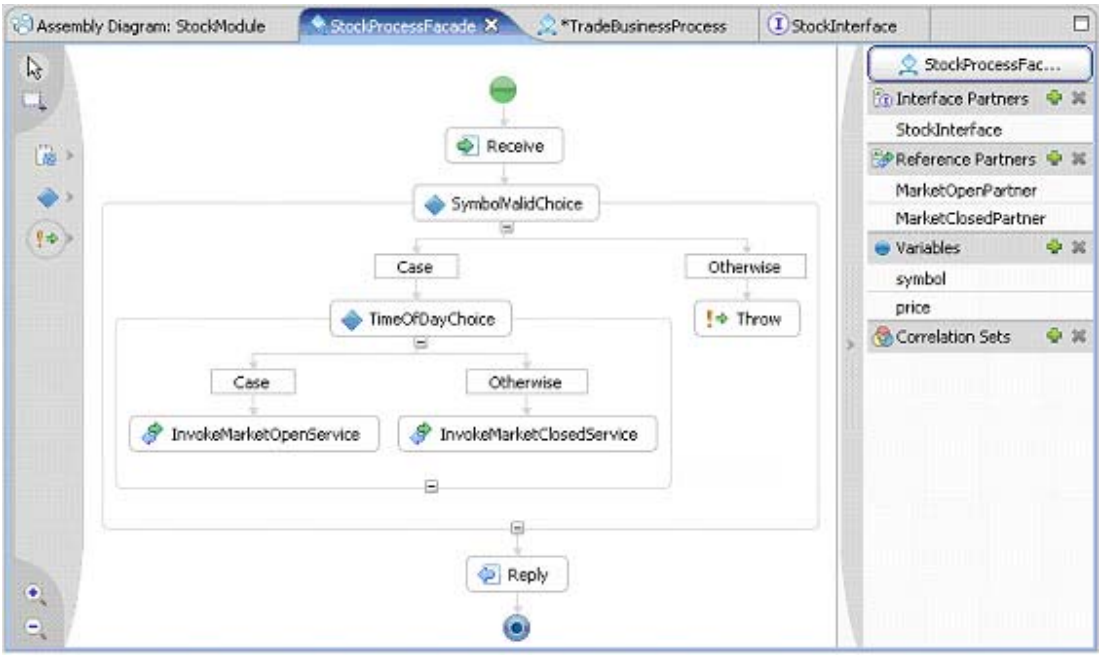
39. Add an **Otherwise** box again from the floating menu above TimeOfDayChoice.

40. Next, we will create an Invoke to an external service. Additionally, the properties of the interface to that service need to be bound to the global variables of the business process (symbol and price). Right-click the **Case** box and select **Add => Invoke**. Rename "Invoke" to `InvokeMarketOpenService`. Select **InvokeMarketOpenService => Properties tab => Browse... => MarketOpenPartner => OK**. Select the **(...)** button next to stockSymbol, then select **symbol => OK**. Select the **(...)** button next to stockPrice and then **price => OK**.

41. Right-click the **Otherwise** box and select **Add => Invoke**. Rename "Invoke" to `InvokeMarketClosedService`. Repeat the steps above to bind the symbol and price variables.

42. The resulting Business Process should be similar to Figure 9.

Figure 9. Business Process



43. Now, test the Business Process component with the Test Component wizard.

Create the client servlet

The following steps walk through the creation of a servlet that calls an SCA component running on WebSphere Process Server. Specifically, it calls a service to calculate a stock price when given the stock symbol.

44. In the Web perspective right-click **Dynamic Web Projects** and select **New => Dynamic Web Project**. Insert `TestModuleWeb` for the name, select **Show Advanced** and change the EAR Project: to `StockModuleApp`. The servlet will not be able to look up the standalone reference if the Web project is not part of the same application. Select **Finish**.

45. Right-click **Deployment Descriptor: TestModuleWeb**, select **New => Servlet...** and insert `StockServlet` for the name. Select **Next**, insert `devworks.example` for the Java package, and then **Finish**.

46. We must include the library project on the build classpath, otherwise the servlet will not have access to StockInterface.java. Select **Dynamic Web Projects**, right-click the **TestModuleWeb** project, select **Properties => Java Build Path => Projects** and check **StockLibrary**.

47. Navigate to **Java Resources => Java Sources => devworks.example**, and then double-click **StockServlet.java**. Replace the doGet() method with the following code:

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        String symbol = request.getParameter("symbol");
        System.out.println("++++++ Calculating price for stock symbol: "+symbol);

        ServiceManager manager = new ServiceManager();
        StockInterface priceService = (StockInterface)
                manager.locateService("StockInterfacePartner");
        try {
                String price = priceService.getStockPrice(symbol);
                response.getWriter().print(price);
        } catch (ServiceRuntimeException sre) {
                response.getWriter().print("Please enter a stock symbol that is three
                        characters long and only composed of letters.");
```

```
                sre.printStackTrace(System.err);
            }
    }
```

48. Press **Ctrl-Shift-O** to organize and resolve imports.

Run the servlet from a Web browser

To deploy the application:

48. Select the **Servers** tab, right-click **WebSphere Process Server V6.0** and select **Start**. When the server is started, right-click the server again and select **Add and remove projects...** to deploy StockModuleApp. Run the application using this address in a Web browser:

    ```
    http://localhost:9080/StockModuleWeb/StockServlet?symbol=IBM
    ```

49. To test other stock symbols, replace "IBM" for the stock symbol as appropriate. To test the fault condition, insert a stock symbol that is longer than three characters and/or is not composed of letters.

Conclusion

In this article, we developed a business process, business state machine, and Java component that is exposed to the outside world with a standalone reference. A servlet was used to invoke the stock price application using SCA's default RMI/IIOP transport. The stock price example described here, while basic, should serve well as an introductory exercise for IBM WebSphere Integration Developer and SCA components.

Resources

- Specifications: Service Component Architecture (SCA) and Service Data Objects (SDO)
- Building SOA solutions with the Service Component Architecture series
- Introduction to IBM SOA Programming Model series
- IBM WebSphere Integration Developer product information
- IBM developerWorks WebSphere Business Integration Zone contains articles, tutorials, code samples, roadmaps, and access to many other resources.

About the author

Robert R. Peterson is part of the enablement team under IBM Software Services for WebSphere. He works to

ensure that the WebSphere portfolio of products brings IBM's clients the greatest value possible. Robert is an accomplished inventor and co-author of WebSphere Application Server V6: Performance and Scalability. He is an alumni of IBM's prestigious Extreme Blue Program and holds a M.S. in Computer Engineering from the University of Florida.